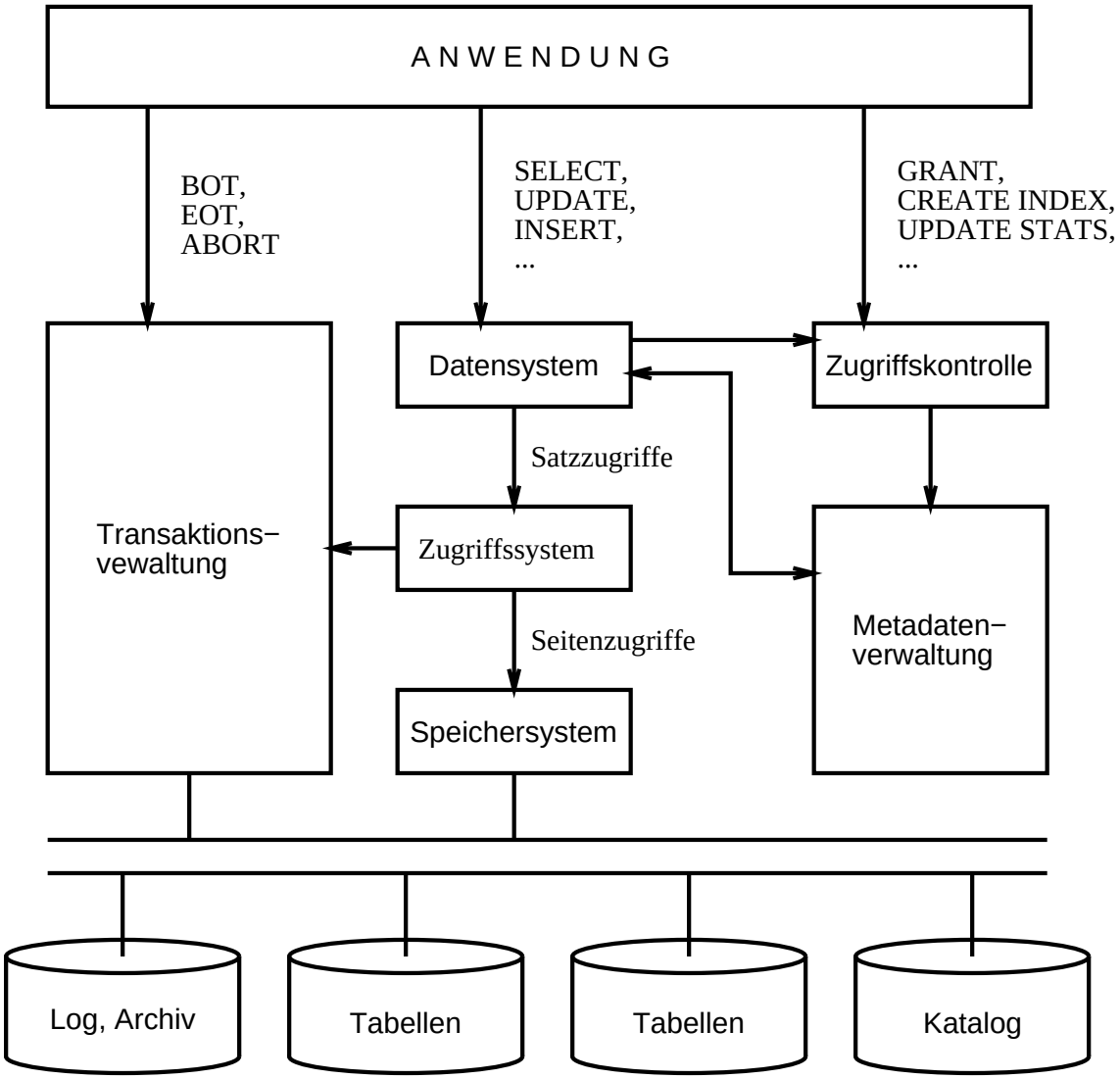
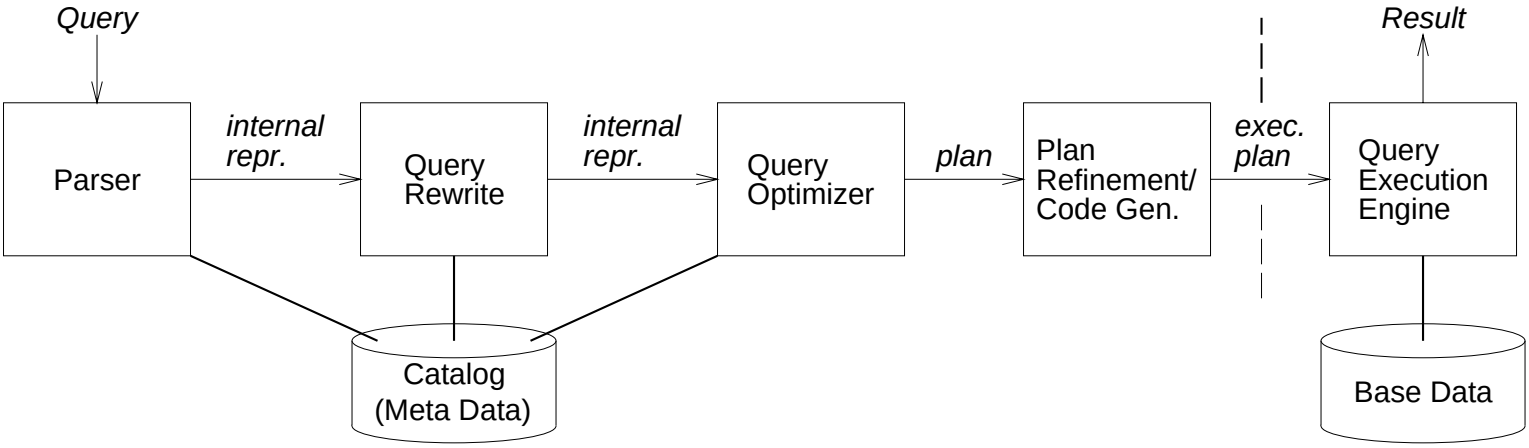
Architektur eines DBMS



Architektur der Anfragebearbeitung (Datensystem)



Query Compiler

- **Parser:** Syntaktische Analyse, Zugriffskontrolle. (Auch semantische Analyse.)
- **Rewrite:** Optimierungen, die gut sind unabhängig vom Zustand der Datenbasis (z.B. Größe von Tabellen, Präsenz von Indexen etc.).
- **Optimizer:** Optimierungen, die vom Zustand der Datenbasis abhängen. Ergebnis ist ein Plan, ein Baum von Operatoren.
- **Code Generierung:** Ausführbaren Plan erzeugen.

Laufzeitsystem

- Interpreter, der auf die Datenbank zugreift.
- Implementierung aller Operatoren.
- Scheduler, der Hauptspeicher für jeden Operator allokiert und entscheidet, welche Pläne zuzulassen sind.

Anfragebearbeitung

- Implementierungsalternativen von Operatoren (z.B. Join, Group-by, Sort, etc.)
- Query Rewrite: Behandlung von Views.
- Optimierung und Kostenmodelle
- Codeerzeugung
- Pufferallokation und Scheduling

Bemerkung

- Wir konzentrieren uns auf read-only Anfragen (i.e., SQL `SELECT` Statements).
- Das ist aber auch genauso für `UPDATE`, `INSERT` und `DELETE` Statements relevant.

Einfache Selektionen

```
SELECT *
FROM Studenten
WHERE semester > 15;
```

Einfacher Table Scan

- Lies sequentiell jede Seite und jeden Slot der Tabelle.
- Gebe Tupel zurück, für die das Prädikat erfüllt ist.

Einfacher Index Scan

- Verwende `Studenten.semester` Index, um die RIDs aller Tupel zu finden, die das Prädikat erfüllen.
- Verwende RIDs, um die kompletten Studententupel zu lesen.

Index Scan mit RID Sort

- Verwende `Studenten.semester` Index, um die RIDs aller Tupel zu finden, die das Prädikat erfüllen.
- Sortiere RIDs nach *page-id*.
- Verwende sortierte RIDs, um die kompletten Studententupel zu lesen.
- **Vorteil:** Keine Random-IO. Geringerer Pufferbedarf.

Einfache Selektionen

Bemerkung

- Damit der Optimierer die richtige Entscheidung treffen kann, werden Kosten für einen kompletten Indexscan im Katalog gespeichert.
- Zusätzlich werden Histogramme gehalten, so daß der Optimierer weiß, wieviel vom Indexscan ausgeführt werden muß.

Index ANDing und ORing

```
SELECT *
FROM Studenten
WHERE semester > 15 AND alter > 30;
```

- Verwende `Studenten.semester` Index, um die RIDs aller Studenten mit `semester > 15` zu finden.
- Verwende `Studenten.alter` Index, um die RIDs aller Studenten mit `alter > 15` zu finden.
- Bilde den Durchschnitt der beiden RID Mengen. (Durchschnittsbildung funktioniert wie Join.)
- Verwende dann die RIDs, um die kompletten Studentupel zu lesen. (Ggf. mit vorheriger Sortierung.)
- Funktioniert natürlich auch mit OR und UNION. Vorsicht, beim UNION muß man Duplikate eliminieren!

Externes Sortieren

```
SELECT *
FROM Studenten
ORDER BY semester DESC;
```

Grundprinzip

- **1. Phase:** Erzeuge sortierte Läufe.
- **2. Phase:** Merge Läufe.

1. Phase:

- Häppchenweise Laden in den Hauptspeicher und Anwendung von Quicksort. Danach Herausschreiben auf Platte.
- Anwendung von *Replacement Selection*. (siehe Knuth, Band 3)

2. Phase:

- Verwende *Priority Queue* bei vielen Läufen.
- Bei sehr, sehr vielen Läufen wird mehrstufiges Mergen notwendig.
- Anzahl der Mergestufen: $\mathcal{O}(log_m n)$
  n Anzahl der Tupel; m Hauptspeichergröße.

Hauptspeicherorganisation in Phase 1 bei Verwendung von Quicksort

- **Grundidee:** Trenne Tupel von der Information, die man zum Sortieren braucht.
- Allokieren einen großen Hauptspeicherbereich.
- Fülle diesen von vorne nach hinten mit Tupeln auf.
- Lege von hinten nach vorne Deskriptoren für die Tupel an.
- Deskriptor: ⟨Zeiger, Sortierschlüssel⟩
- Das macht man, damit man nur die Deskriptoren zu sortieren braucht.
  - Spart Kopierkosten beim Sortieren, da die Deskriptoren i.a. kleiner als die Tupel sind.
  - Funktioniert auch, wenn die Tupel unterschiedlich groß sind, da die Deskriptoren alle gleich groß sind.
- Ein ähnliches Prinzip ist bei der Implementierung von sehr vielen Operatoren anwendbar.

Sortierung

Alternative

- Verwende einen B-Baum zum Lesen der Tabelle.
- Nur effizient, wenn der B-Baum geclustert ist.
- Ansonsten gibt es schrecklich viel Random IO.
- Wiederum verwendet der Optimierer die Kosten für den kompletten Scan, um die Entscheidung zu treffen.

Joinmethoden

```
SELECT *
FROM Emp e, Dept d
WHERE e.dno=d.dno AND
e.salary>d.budget/10;
```

Überblick

- Nested-loop Join
- Index Nested-loop Join
- Merge Join
- Grace Hash Join
- Hybrid Hash Join
- Double-pipelined Hash Join
- Order-preserving Hash Joins

Bemerkung

- Wir unterscheiden zwischen *innerer* und *äußerer* Tabelle: $A \bowtie B$ vs. $B \bowtie A$.
- Das ändert nichts am Ergebnis aber an den Kosten.
- Bei manchen Methoden ist unter gewissen Umständen nur eine Richtung möglich: z.B. bei Index Nested-loop Joins und bei Order-preserving Hash Joins.

Nested-loop Join

Algorithmus

- Für jedes Tupel der äußeren Tabelle:
  - Scanne durch die innere Tabelle.
  - Führe für jedes Tupel der inneren Tabelle das Joinprädikat aus.
  - Bei Erfolg, liefere Tupelpaar zurück.

Bewertung

- Quadratischer CPU Aufwand.
- IO okay, falls innere Tabelle in den Hauptspeicher paßt. Ansonsten furchtbar hohe IO Kosten.
- Universell für jede Art von Joinprädikat einsetzbar.

Index Nested-loop Join

Algorithmus

- Voraussetzung: Innere Tabelle hat einen Index mit dem man einen Teil des Joinprädikates (in KNF) ausführen kann.
- Für jedes Tupel der äußeren Tabelle:
  - Verwende den Index, um matchende RIDs der inneren Tabelle zu finden.
  - Lese die matchenden Tupel der inneren Tabelle anhand ihrer RIDs.
  - Wende weitere Joinprädikate an und liefere Ergebnistupel zurück.

Bewertung

- Nur gut, wenn die äußere Tabelle sehr klein und die Innere sehr groß ist.
- Natürllich nur einsetzbar, wenn es einen passenden Index gibt.

Index Nested-loop Join

Variante 1

- Sortiere äußere Tabelle nach Joinattribut(en), damit der Indexlookup schneller wird.
- Reduziert Kosten beim Indexzugriff.
- Erlaubt gleichzeitige Bearbeitung mehrere Tupel der äußeren Tabelle.
- Löst aber nicht das Random IO Problem beim Zugriff auf die Tupel der inneren Tabelle mit RIDs.

Variante 2

- Sortiere RIDs vor jedem Zugriff auf die matchenden Tupel der inneren Tabelle.
- Bringt in der Regel nichts, weil es sich eh nur um sehr wenige matchende Tupel pro äußeres Tupel handelt.

Merge Join

Prinzip

- Beide Tabellen seien gemäß eines der Joinprädikate sortiert.
- Bearbeite die ersten Tupel der beiden Tabellen. Je nach Joinprädikat werden Ergebnispaare ausgegeben und weitergeschaltet (Merge). Andere (sekundäre) Joinprädikate werden natürlich bei ebenfalls beachtet.
- Die Details des Algorithmus sind ein wenig fieselig abhängig davon wie man die Sortierung erreicht.

Merge Join

Wie erhält man die Sortierung?

- Durch einen (geclusterten Index)
  - Kann man Lesen der Tupel anhand der RIDs evtl. auf nach dem Join vertagen?
  - RID Sortierung kann wiederum interessant werden.
- Durch explizite (externe) Sortierung:
  - In diesem Fall führt man den Merge der einzelnen Läufe und den Merge mit der anderen Tabelle gleichzeitig durch.
  - D.h. man benötigt im Merge Join Operator eine Priority Queue.
- Sortierung besteht noch von einem vorherigen Join. (Dies ist ein ganz wichtiger Aspekt des Mergejoins.)
- Kombinationen sind natürlich möglich: z.B. Sortierung einer Tabelle durch Index, Sortierung der anderen durch explizite Sortierung.

Classic Hash Join

Algorithmus

- Lese die innere Tabelle in den virtuellen Speicher.
- Baue eine Hashtabelle im virtuellen Speicher auf die innere Tabelle.
- Scanne durch die äußere Tabelle und verwende die Hashtabelle, um matchende Tupel zu finden.
- Wende weiteren (sekundären) Joinprädikate an und gebe Ergebnispaare aus.

Bewertung

- Klappt nur für Equijoins. (Das sind zum Glück 90% aller Joins.)
- Furchtbar, falls die innere Tabelle sehr viel größer als der Hauptspeicher ist. Dann Random-IO durch Swappen des Betriebssystems.
- Außerdem könnten sich andere Prozesse und Benutzer auf dem Rechner beschweren.
- Macht kein kommerzielles System so.

Blockwise Hashed Nested-loop Join

Algorithmus

- Lese einen hauptspeichergroßen Block von Tupeln der äußeren Tabelle.
- Baue eine Hashtabelle auf diesen Block von Tupel auf.
- Lese innere Tabelle vollständig und verwende die Hashtabelle, um für jedes innere Tupel die matchenden äußeren Tupel zu finden.
- Wiederhole diese Schritte für jeden Block von Tupeln der äußeren Tabelle.

Bewertung

- Funktioniert in vielen Fällen erstaunlich gut.
- Allerdings ist diese Methode nicht gut, wenn beide Tabellen sehr groß sind und wenig Hauptspeicher zur Verfügung steht. I.e., wenn es viele Iterationen gibt.
- Funktioniert wie alle hashbasierten Methoden nur bei Equijoins.

Grace Hash Join

Algorithmus

- Wende eine Partitionierungsfunktion auf das Joinattribut der inneren Tabelle an. Ziel ist es, daß jede Partition in den Hauptspeicher paßt. Ggf. muß man rekursiv partitionieren. (Schieflage der Daten oder sehr kleiner Hauptspeicher.)
- Partioniere die äußere Tabelle unter Verwendung derselben Partitionierungsfunktion. (Berücksichtige auch rekursive Partitionierungen.)
- Wende einen Classic Hashjoin paarweise auf die Partitionen an.

Einige Tricks

- Bloom Filter (Bitmaps) zum Vorfiltern der äußeren Tabelle verwenden.
- Rolle der inneren und äußeren Tabelle für manche Tabellen tauschen.
- Partitionen zu einer zusammenfassen, falls möglich.

Grace Hash Join

Bewertung

- Verwende stets die kleinere Tabelle als Innere.
- Sehr häufig der Sieger unter den klassischen Joinmethoden. In der Regel kommt man mit einmal Schreiben und Lesen der beiden Tabellen aus. Es sind kaum Seeks notwendig.
- Wiederum nur bei Equijoins einsetzbar.

Hybrid Hash Join

Grundidee

- Man fängt an wie im Classic Hash Join. I.e., Aufbau einer Hauptspeicherhashtabelle für die innere Tabelle.
- Wenn Hauptspeicher voll, dann fängt man die innere Tabelle zu partitionieren.
- Eine Partition der inneren Tabelle bleibt allerdings im Hauptspeicher (mit Hashtabelle).
- Äußere Tabelle wird partitioniert, wobei erste Partition der äußeren Tabelle gleich mit der ersten Partition der inneren Tabelle gejoint wird.

Bemerkung

- Es klappen im Wesentlichen dieselben Tricks wie beim Grace Hash Join.

Hybrid Hash Join

Bewertung

- Adaptiv: Wenn man vorher nicht weiß, ob innere Tabelle in den Hauptspeicher paßt, dann funktioniert der Hybrid Hash Join wie der Classic Hash Join im günstigen Fall und wie der Grace Hash Join im ungünstigen Fall.
- Im ungünstigen Fall sogar weniger IO als der Grace Hash Join, weil die erste Partition beider Tabellen nicht auf Platte geschrieben werden muß.
- Allerdings: Gewinne im Vergleich zum Grace Hash Join nur dann bedeutsam, wenn der Hauptspeicher fast so groß wie die innere Tabelle ist.
- Außerdem kann der Hybrid Hash Join zu mehr Seeks als der Grace Hash Join führen. Wieso?

Mehrwege Join Anfragen

- Plan mit Baum von Joins.
- Joinmethoden können beliebig kombiniert werden.
- Vorsicht allerdings: Bei Nested-loop Joins muß innere Tabelle stets materialisiert sein.
- Vorsicht: Manche Joinmethoden sind Pipelinebreaker, manche nicht. Dies macht sich auf den Hauptspeicherbedarf des Plans und die IO-Kosten bemerkbar.
- Beachte, daß man versucht das Schreiben von Zwischenergebnissen auf Platte soweit wie möglich zu vermeiden.

Iterator Modell

Prinzip

- Generell können alle Operatoren beliebig zusammengestöpselt werden und dann ausgeführt werden, da alle Operatoren dieselbe Schnittstelle haben:
  - **open:** Führt vorbereitende Arbeiten (z.B. Allokieren von Speicher) aus.
  - **next:** Liefert das nächste Tupel.
  - **close:** Räumt auf (z.B. Freigabe des allokierten Hauptspeichers).

  So kann ein Operator einen beliebig anderen Operator aufrufen, um Zwischenergebnisse zu bekommen. Endergebnis der Anfrage wird durch Aufruf des Wurzeloperators des Baumes erzielt.
- Besonderheiten der Implementierung der Operatoren (wie z.B. Pipelining oder Pipelinebreaking) werden verkapselt.
- Der Optimierer muß allerdings von allen Besonderheiten wissen, um einen guten Plan erzeugen zu können.

Double-pipelined Hash Join

Algorithmus

- Baue parallel Hashtabellen im Hauptspeicher für beide Tabellen auf.
- Vorgehensweise für ein Tupel der äußeren Tabelle:
  - Verwende Hashtabelle für die innere Tabelle, um alle (bisherigen) Matches zu finden.
  - Trage das Tupel in die Hashtabelle für die äußere Tabelle, damit Matches mit späteren Tupeln der inneren Tabelle gefunden werden.
- Analog für ein Tupel der inneren Tabelle.

Bewertung

- Besonders gut, wenn es darauf ankommt sehr schnell Ergebnisse zu erzielen. I.e., wenn das gesamte Ergebnis nicht unbedingt gebraucht wird.
- Besonders gut in verteilten und parallelen Systemen, um Parallelität durch Fließbandverarbeitung auszunutzen.
- Besondere Maßnahmen sind notwendig, wenn der Hauptspeicher voll ist. (Siehe Übung.)

Order-preserving Hash Joins

- Funktioniert wie Grace Hash Join.
- Erhält allerdings die Sortierung auf der äußeren Tabelle bei.
- Kein Gewinn in der IO. Allerdings erspart es CPU Kosten für die Sortierung.
- Außerdem geringerer Hauptspeicherbedarf.
- Macht weitere Optimierungen möglich/attraktiv. (Anwendung von RID Joins.)
- Überkreuzmergen bei Mehrwegejoins.
- Lit.: Claußen, Kemper, Kossmann: MIP 1998.

RID Joins

```
SELECT *
FROM Studenten s, Hören h, Dozent d
WHERE s.name=h.stud AND h.doz=d.name;
```

Vorgehensweise

- Lese kompletten `Stundent.name` Index, um ⟨Name, RID⟩ zu erhalten.
- Führe ⟨Name, RID⟩ ⋈ Hören mit einer herkömmlichen Joinmethode aus. Das ist billiger, weil ⟨Name, RID⟩ viel kleiner als die komplette Studententabelle ist.
- Führe dann Join mit den Dozenten aus. Wiederum billiger als gewöhnlicher Join, weil das Zwischenergebnis auch kleiner ist.
- Hole dann die kompletten Studententupel anhand der RIDs. Evtl. die RIDs sortieren, damit das billiger wird.

RID Joins

Bewertung

- Im allgemeinen nur gut, wenn das Endergebnis sehr klein ist; i.e., die Selektivität der Joinprädikate sehr hoch ist oder zusätzliche Prädikate die Ergebnismenge einschränken; z.B.

  ```
  d.fak = 'Informatik'
  ```

- Funktioniert allerdings gut in Kombination mit Order-preserving Hash Joins, da man sich dann die Sortierung der RIDs am Ende sparen kann.

Semi Join Programme

$$\text{text}B\ \text{text}A\ \text{text}\bowtie$$
$$\text{text}C\ \text{text}A\ \text{text}\ \text{text}\bowtie$$

- Funktioniert sehr gut, wenn A klein ist und Joins mit A sehr selektiv sind. Dann sind alle Joins und Semi-Joins mit A sehr billig und der letzte Join wird stark verbilligt.
- Maximaler Gewinn im zentralen Fall: ca. Faktor 2.
- Im verteilten Fall sind die Gewinne bei solchen Plänen potentiell höher. Durch Ausnutzung von Parallelität oder durch Einsparung von Kommunikationskosten.

Gruppierung

Überblick

- Nested-loops (klar)
- Index Nested-loops (klar)
- Durch Sortierung (klar)
- Hashing, Hybrid Hashing (klar)
- Early Aggregation (bei Sortierung oder durch Hashing)

Bemerkung

- Eine Gruppierung ist (fast) wie ein Selfjoin.
- Duplikatelimination ist eine besondere Art von Gruppierung.

Early Aggregation

Grundidee

- Tupel mit dem gleichen Gruppierungsschlüssel kann während der Verarbeitung sofort zusammenfassen.
- Beim Gruppieren durch Sortierung erhält man hierdurch kürzere Läufe. Tupel mit gleichem Gruppierungsschlüssel können beim Sortieren trivialerweise sofort erkannt werden.
- Beim Gruppieren durch Partitionieren erhält man hierdurch kleinere Partitionen. Hierbei braucht man eine separate Hashtabelle um Tupel mit gleichem Gruppierungsschlüssel zu erkennen.

Subtile Beobachtung

- Early Aggregation ist oft besonders effektiv, wenn die Gruppierung nach einem Join erfolgt, da durch den Join bereits eine gewisse Clusterung der Tupel erfolgt ist.
- Beispiel:

```
SELECT d.alter, count(*)
FROM Student s, Hören h, Dozent d
WHERE s.name=h.stud AND h.doz=d.name
GROUP BY d.alter;
```